Multi-Platform Software Development Using the Model-View-Controller Design Paradigm

Mark Wittenberg, Novell, Inc.

Abstract

Producing a program for more than one platform (such as Macintosh and Windows) is difficult and resource expensive, and it is difficult to maintain functional equivalence between the versions. Engineering, quality assurance, documentation, and maintenance efforts increase in proportion to the number of supported platforms. Programs with a Graphical User Interface (GUI) make this situation worse. To solve this problem we present an architecture that reduces the engineering work required to build multi-platform applications, guarantees functional equivalence between platforms, encourages users to form the same conceptual model of the application on all platforms, provides a firm basis for implementing scripting, allows a native GUI for each platform, and simplifies inter-application communication across platforms. The architecture merges the Model-View-Controller and the Client-Server paradigms.

Contents

- Introduction
- Motivation
- Solution — Distributed MVC Architecture
- Enhanced Client-Server Architecture
- Benefits
- The Conceptual Model
- The Model-View-Controller Triad
- Distributed MVC Revisited
- Scripting
- Functional Specification
- User Interface Specifications
- Testing
- Remote Debugging
- Client-Server Communications
- Peer-to-Peer Communications
- Single-Platform Development
- Limitations
- Summary
- Acknowledgements
- Bibliography

Introduction

It is an increasingly common requirement that a company provide substantially the same application on several platforms; Windows and Macintosh are common today, and Presentation Manager and the X Window System are gaining popularity. DOS does not provide much in the way of graphical interfaces, but there are a very large number of DOS machines in the world today, so applications that are usable in a non-graphical environment can substantially increase their market by providing a DOS version.

Writing an application that is completely portable between any two of these platforms, let alone all of them, is a formidable task, especially if one desires to provide a state-of-the-art user interface on each one. On the other hand, rewriting the application for each platform is expensive and error prone. In addition, a developer is subject to implementing subtly (or not so subtly) different applications on each platform.

A successful product needs to be adequately tested and documented; implementing an application from scratch for each platform requires that it be retested as well, and increases the chance that the documentation will have to be redone completely. All of this is not only expensive, but also increases the time it takes to get the product to market.

Motivation

Applications that run well on a number of platforms enjoy a large potential market, but they may not be able to exploit that market unless they can be brought quickly to market, at a reasonable cost.

Several studies have shown [Mey 90] that the major software expense is not developing a product, but in maintaining it. Maintenance consists of fixing bugs,

adapting to new requirements, adding features, and porting to new platforms.

Designing for portability and improvability drastically reduces maintenance costs; only part of the application needs to be changed when it is time to release the next -version of your product, or to release the current version on a new platform.

Building an application that compiles and runs unchanged on every platform is extremely difficult; the organizational issues, such as maintaining a single copy of the source across platform-specific develop-ment teams, are as difficult as the technical ones.

In many applications it makes sense for multiple users to share access to the same data; if so, you need to provide shared access from different platforms. You might want to support real-time updates of views of shared data; that is, if several users are viewing the same data simultaneously, all of them would immediately and automatically see any change made by any one of them.

Summary of major needs:

- Support the application on all popular platforms.

- Allow data to be shared concurrently from all platforms and provide real-time updating of all views of shared data.

- Ensure non-duplication of effort by testing and documentation.

- Ensure that the applications share the same *conceptual model.* The conceptual model is the user's explanation for how the program works.

Solution — Distributed MVC Architecture

The architecture we present here does not solve all of the problems we listed, but it does give a framework for working on them.
We have combined the *Client-Server* architecture with the *Model-View-Controller* paradigm, and explicitly recognize the importance of the *Conceptual Model;* we call it the *Distributed MVC* architecture. A major accomplishment of this architecture is ensuring the integrity of the system's conceptual model across implementation platforms. Later sections will explain the Model-View-Controller paradigm and Conceptual Models in detail.

Distributed MVC separates the user interface from the application semantics to facilitate implementing multiple user interfaces; it does this by passing messages between the user interface on the client and the semantic model on the server.

A Client-Server architecture splits an application into a *server* which provides a *service* to a number of *clients*. Each client connects to the server, issues service requests, and receives responses. Clients and Servers typically reside on different machines, although this is not a requirement. The Client-Server architecture allows the (presumably) resource-intensive server to run on a large fast computer while allowing the clients to run on smaller, more accessible workstations or departmental computers. It also allows centralized management of the server, a useful setup if a large number of clients are involved. Typical Client-Server programs are Electronic Mail systems and Database systems.

The Model-View-Controller paradigm decomposes a task into a Model, which implements the semantics of the task, a View, which displays the state of the model to the user, and a Controller, which the user manipulates to modify the state of the model. Our adaptation splits the application into a Client and a Server, moves the Model to the Server and adds a Client-Model to provide the same abstraction of a Model to the Views and Controllers.

Enhanced Client-Server Architecture

What does the Distributed MVC architecture add to the Client-Server architecture?

Distributed MVC provides a higher level of abstraction to the client: the server is tailor-made to the specification of the client. Database Management System servers, for example, provide generic data base services; application-specific semantics (such as the definition of tables and the semantics of updates) are left to the application. In a Distributed MVC design, the server would be application-specific; for example, a Personnel Database server. The Personnel Database is itself a client of a DBMS, but presents a higher level of abstraction to the rest of the program.

Coad and Yourdon [Coa 91] divide the design areas into four components, shown in the center row of the Figure 1; the upper and lower rows show the location of those components (client or server) in the Client-

Server architecture and in our Distributed MVC architecture. Because the conceptual model is so important, we move the Problem Domain Component (that is, the part that implements the application-specific semantics) from the client (in the Client-Server architecture) to the server. This sharing of the problem domain ensures that the Design Model is the same on all platforms; the user's Conceptual Model should therefore be the same as well.

By splitting the user interface into the view and controller components, we make scripting and testing easier, and make remote debugging simpler. Scripting, Testing, and Remote Debugging are discussed later.

Client-Server Design Compon

Client User Interface	Client Applicatio Semantics	Server Connection Tasks	Server DBMS
Human Interactio Componen	Problem Domain Componen	Task Managemer Component	Data Managemer Component
Client View Controller	Server Model	Server Connection Tasks	Server DBMS

Distributed MVC Compone

Mapping Distributed MVC Components to

Figure 1

Benefits

- We ensure that all of the applications are functionally equivalent, as the Model implements all of the functionality of the program. We don't guarantee that the user interfaces are equally usable, only that all the functionality of the system resides in the Model.

- Testability is enhanced because the functionality needs to be tested only once. Of course, the user interfaces must still be tested on each platform. We'll have more to say about testing later.

- Usability is enhanced because the user will form the same conceptual model as to how the application works, regardless of the platform being used.

- Documentation is made somewhat easier, as it can be separated (logically, if not actually) into **Functionality** and **How-to-Use** sections. The Functionality section is the same for all platforms.

- We provide a firm basis for implementing *scripting.* Because the Model implements all of the application's functionality, and is accessible only by passing requests and receiving replies, scripts are implemented simply by saving and replaying the message streams flowing into and out of the server.

- We can provide real-time updating of shared views by adding the views as *dependents* of the model; if the model changes, it will send its dependents an update message, and the views can query the model for the new data. This mechanism works transparently for all client platforms.

- The program designer is forced to separate the design into two parts: the **Functional Specification** which describes the Model, and the **User Interface Specifications** (one or more per platform), which describe the user interfaces. This two-part design helps to separate **what** the program does from **how** the user makes it do that, which is an aid in implementation, debugging, and technical support. An important implication is that a bug that appears in only one implementation must be in the View-Controller code on that platform; it can't be in the Model code, which runs on the server.

- The View-Controller code for each platform can be done by separate teams at the same time. There is no need to wait for the first implementation and then have each team "port" the code to their platform. That is, people need not try to discover for themselves what part of the code implements the functionality, port that, and rewrite the user interface.

The Conceptual Model

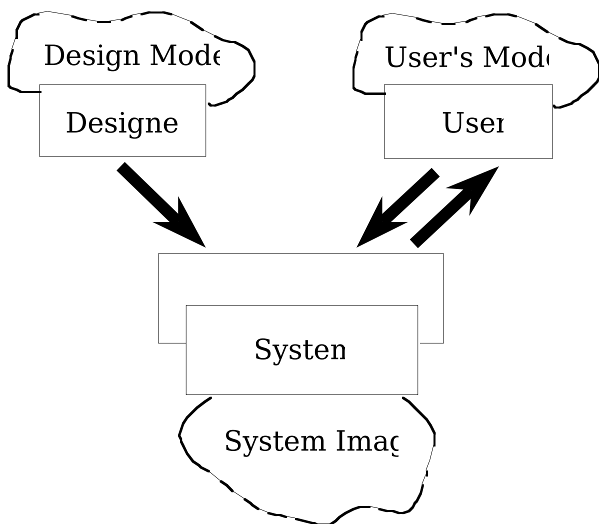
A conceptual model[1] is a person's explanation for how a thing works. Donald Norman says "These models are essential in helping us understand our experiences, predict the outcomes of our actions, and handle unexpected occurrences. We base our models on whatever knowledge we have, real or imaginary, naive or sophisticated … The real point … is that everyone forms theories (mental models) to explain what they have observed."

Norman points out that there are three conceptual models: the *design model,* the *user's model,* and the *system image.* See Figure 2.

"The design model is the conceptualization that the designer has in mind. The user's model is what the user develops to explain the operation of the system. … [The] user and designer communicate only through the system itself: its physical appearance, its operation, the way it responds, and the manuals and instructions that accompany it. Thus the *system image* is critical: the designer must ensure that everything about the

[1]See [Nor 88].

product is consistent with and exemplifies the operation of the proper conceptual model.”[2]



Three Aspects of Mental Mo

Figure 2

The Model-View-Controller Triad

MVC[3] is a problem decomposition paradigm, originally developed in Smalltalk-80. In it, the designer factors an application into three pieces:

- **Model**: the part that represents the model of the underlying application problem domain.
- **View**: the part that presents the model to the user.
- **Controller**: the part that allows the user to modify the model.

The three pieces communicate by sending messages. The Controller sends messages to the Model to effect a change; the Model sends messages to the View to change the display. Adopting the point of view of the Model, we shall call the message stream flowing to the Model the *Input Stream,* and the message stream flowing from the Model the *Output Stream.*

In practice, the Controller may also send messages to the View (such as a message to change window size), the View may send messages to the Model (such as a message asking for the current state of the model), and the Model may send messages to the Controller (such as a message that its state has changed).

The Model generally maintains a list of *dependents,* who should be notified when it changes state. The View is always a dependent, and the Controller and other Models may also be dependents. The list of dependents is just a list; ideally the Model knows nothing about its dependents except that they need to be notified when the Model has changed state.

Figure 3 shows the main MVC components. The heavy black lines indicate the major message flows, the dashed lines indicate the minor message flows, and the gray lines indicate attachments to either physical devices or software components outside of the design domain, such as an operating system.

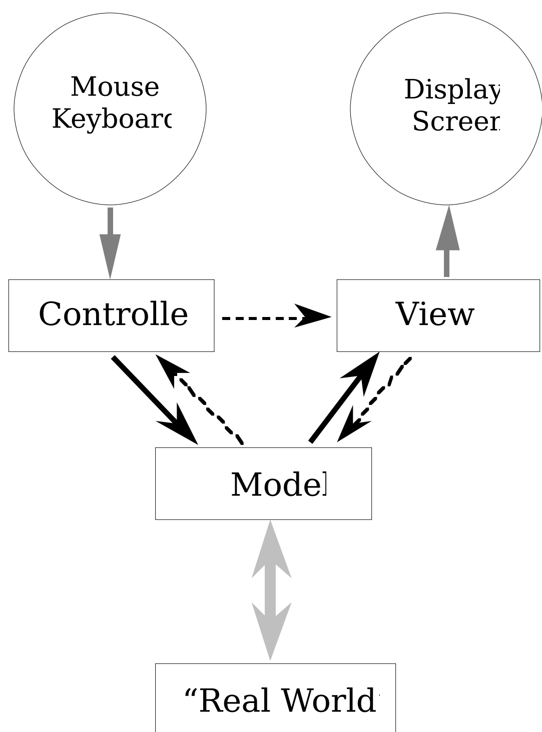
In Smalltalk-80 the programmer designs Classes to implement the MVC triad and instantiates Objects to effect them. Typically there will be a number of MVC triads in an application, corresponding to the number of “application domains” encompassed. For example, a CAD system might have a document editing triad, a memo (text) editing triad, a database (filing) triad, a design-consistency checking triad.

Although decomposing an application into a number of Models is useful, we don’t restrict ourselves to an Object-Oriented design in this paper. For simplicity,

[2]The diagram and the quotes in this section come from [Nor 88], because I couldn’t have said it better myself.
[3]Borrowed from Smalltalk. A good description can be found in [Kra 88].

we will mainly address the single-model case. However, the architecture can be extended to the multiple-model case.

It is important to note that there may be several View-Controller pairs associated with one Model. A straightforward example is in a text editor. Some people like keyboard commands and some people like mouse-based commands. Some people[4] like some of each, depending on the function to be performed. This situation is easy to handle with a text-editor Model, a text-editor View, and two Controllers: a keyboard Controller and a mouse Controller. Either one or both Controllers can be active and sending commands to the Model; the Model sends the results to the View to be displayed. In this case the two View-Controller pairs share the same View, but this need not be the case.



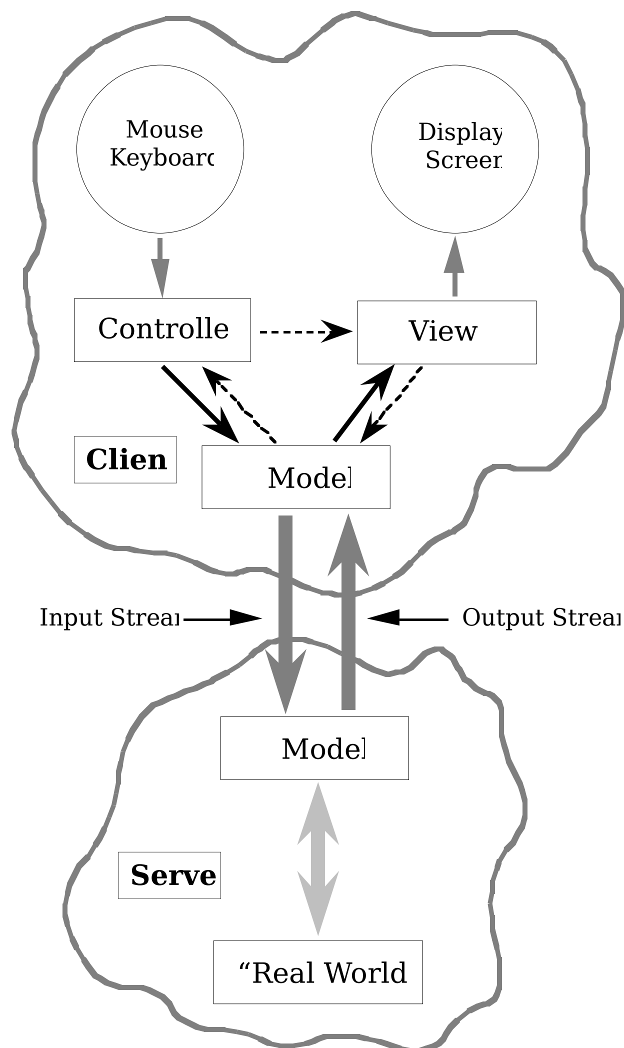
Model-View-Controller Desi

Figure 3

[4]Me, for example.

Distributed MVC Revisited

We can now explain the Distributed MVC architecture more fully. Figure 4 shows the application split into the client and server pieces. The Model has been moved to the server to ensure the integrity of the system's conceptual model, and a new Model has been created in the client to implement the system's semantics by passing requests to the server model. Each client's model maintains a local list of dependents, and is itself a dependent of the server model.



Distributed MVC Desi

Figure 4

Distributed MVC is transparent to the Views and Controllers; the client Model transparently passes View and Controller messages to the server. In particular, the Controllers and Views do not maintain any notion of the network address of the server, as they are not aware that communication is taking place over a network. Of course, the client's Model must somehow find the server at initialization, and there is presumably some sort of user interface for specifying a server. Therefore, one View-Controller pair knows about servers, but it will not participate in the main function of the application.

In Distributed MVC, the Model defines the System Image and the way the System Image responds. Thus, half of this critical component of the user's understanding of the application is shared with the other implementations. The other half – the documentation and the physical appearance – is not shared (although part of the documentation can be), but the separation of the user interface from the functionality helps to focus the designer's attention on the part particular to each platform.

This architecture:

- Implements the functionality of the application once, in the server.
- Presents a uniform programming interface to every client (that is, to the client's Model).
- Separates the design of the user interfaces from that of the functionality.
- Divides the user interface into display and manipulation components and allows them to be interchanged.
- Reduces order dependencies in the development process. See the Testing section for details.

Scripting

Scripting is the ability to record and playback a series of commands, and to record the results. It differs from a macro facility[5] by recording commands, not keystrokes and mouse clicks, and by recording the results as well.

For example, a script command might be Send "Get Info" to the current selection, whereas the equivalent macro recording might be Click at screen position {h=67, v=17} and drag to {h=71,v=78}. The reader (of the macro file) must already know that "Get Info" is a menu item somewhere near the described location.

Macros are notorious for being highly dependent on the size of menu titles, the position of items in a menu, the size of the screen, the number of items drawn in the window, the speed of the CPU, the phase of the moon, and so on. As such, they are completely non-portable between platforms (indeed, they are generally non-portable between versions on the same platform) and are extremely hard to read or modify.

Macros also cannot test functionality separately from the user interface, and so cannot discriminate user interface bugs from functional bugs. Also, they cannot be used in regression tests[6] to detect bugs automatically, as they do not capture output When running a test, something should probably happen as a result of executing a command, but what? In the macro version the tester has no way of knowing. In the script version, a little later there should be a command describing the expected response, such as Get "Size is 103,476 bytes, Location is Ann Arbor." A simple utility to print the differences in a pair of text files suffices to automatically detect bugs.

Either the input or the output stream can be recorded (a **Tap**), synthesized (an **Injection**), or diverted (a **Siphon**). There are four useful configurations of the message streams, shown in Figure 5:

Type	**Model Input**	**Model Output**
User	Live	Live
Replay	Inject	Live
Record	Tap	Tap
Batch	Inject	Siphon

Figure 5

The application is normally run in **User** mode, but the user will sometimes wish to **Record** his or her actions and later **Replay** them. **Batch** operations would normally be used for regression testing, but for many applications batch operations would be a useful user's mode as well. A slight modification also allows remote debugging (see the **Remote Debugging** section).

Functional Specification

A functional specification describes what the application does, not how it does it or how the user interacts with it. If you have formalized your definition of the product (perhaps with Abstract Data Types,[7] Push-Down Automata,[8] or an axiomatic system), then both implementing the model and writing a test suite is simplified.

In the real world, such formal product specifications are not common, but the MVC decomposition provides a reasonable alternative: specify every message that can be sent to the model and the set of possible responses. The set of legal sequences of messages defines a language, so you should consider writing a formal definition of that language[9] (i.e., a grammar). This is not a complete formal definition of the application, of course, since the semantics are missing. However, it is sufficient for the quality assurance department to generate a test plan.

The specification of the set of messages and responses is necessary whether or not a more formal specification

[5] That is, a facility that records the user's actions, not one in which the user writes macros explicitly – a "Read My Lips" tool.
[6] Regression testing is the process of repeating a subset of tests after fixing previously discovered bugs. The tests that revealed the bugs and tests that apply to functions affected by the fixes are repeated.
[7] A good definition can be found in [Mey 90].
[8] See [Aho 72], for example
[9] See your favorite language theory text, such as [Aho 72] or [Aho 86].

is provided; the messages are the relevant entities for implementation, documentation, and testing. If you provide a more formal specification, someone should verify the message specification against the formal specification.

Note that the messages should not contain any text; this requirement is important in order to support the international market. Your application should support clients in different languages simultaneously, which implies that only the client user interfaces should generate text.

User Interface Specifications

There should be a separate user interface specification for each user interface, detailing how the user issues commands and sees results. The first specification should be for the scripting language so that QA can start writing tests.

The GUI specifications should specify which actions in the Controller generate which messages, and should detail the actions of the View receiving each message. For clarity, you may want to repeat the semantic intent of each Controller message (already described in the Functional Specification) However, the message stream constitutes the most important information required by both the programming team and the QA team.

The technical documentation department has different requirements, because the message stream is not significant to the user except for providing an explicit description of the conceptual model supported by the application. Many user's manuals benefit from a **Theory of Operation** section, which can derive directly from the message definitions.

If the scripting facility is available to customers, then the script language should be documented with a formal language definition so the documentation team can accurately describe it.

Testing

Let's assume that your application needs to run on DOS, Windows, OS/2, Macintosh, and the X Window System. How are you going to test all five versions? In our architecture, you first write the Model, running on the server, and then write the View-Controller pairs, running on the clients.

The first View-Controller pair to implement is the scripting mechanism. The View receives a stream of messages and generates a human-readable text describing them. The Controller takes text describing a stream of messages (in the same language that the View generates), parses it, and sends the resulting messages to the Model.

In the meantime, Quality Assurance can start writing test scripts. As soon as this first MVC triad is complete, testing can begin, while engineering goes on to write the "real" Graphical User Interfaces (GUIs), and of course, to fix the bugs in the model that are found by QA.

One of the interesting benefits to this architecture is that QA can test a GUI piecemeal: the scripting controller and the model can be hooked up to a GUI view; you can type commands into the controller and examine the behavior of the view.

Conversely a GUI controller can be hooked up to a scripting view, and be debugged by examining the output messages.

In addition to providing the user with a batch facility, scripting also provides a platform-independent test bed. The logical separation of the View from the Controller allows us to feed the Controller portion of a script into the Model, but hook the output stream to a live View. We can then visually verify the appearance of the View.[10] Any platform with a working Controller can generate the input script, so long as there is also a working model. Unlike macros, scripts are platform-independent: a script generated on one platform can be used on any other platform unchanged.

Scripting also allows easy regression testing of the Model; given valid input and output scripts for a test, all that a regression test requires is to feed in the input script, save the output stream, and compare the new output stream with the saved known good one. If they are the same, all client applications that have not been

[10] A speed control on the Script Controller would be useful.

modified are known to be correct, and do not need retesting.

Regression testing of the Controller and of the View is harder. In the former case you need to generate the same user actions. and compare the output stream with the saved output. Macros could be useful here, in spite of their problems. In the latter case, you feed the saved input (i.e., the *input stream*) to the View and visually compare the results.

Remote Debugging

We have shown how to use the message stream to design a batch system (scripting) and a graphical user interface. The message stream can also provide remote debugging. By tapping into the output stream of a remote customer, you can see the symptoms the user encounters; by tapping into his or her input stream you can test the system, and the customer can see the results at the same time. Remote debugging is possible even if you are running on one platform and your customer is running on another. Of course, in this case, if you don't observe the bug that the customer is encountering, then you've just localized the bug to the user interface of the client's platform. Supporting dial-in lines may be a good idea for this reason even if you think that dial-in lines are too slow for normal use. You should provide a security mechanism so that your customers can control who can tap into their systems.

Similar functionality is provided by programs such as Timbuktu[11] for the Macintosh and **Carbon Copy**[12] for DOS, although they work differently. These programs work by emulating the target workstation's display device at the controlling workstation's display, and passing input device actions from the controlling workstation into the target workstation.

While useful, they suffer from several generic problems:

- The "messages" being passed are at a much lower level of abstraction than are the Distributed MVC messages, including mouse movements as well as typing corrections. Therefore, they do not allow a controlling workstation to be of a different kind than the target. They can also suffer from slow response time, due to the amount of unimportant data they must transmit.

- They can't save the meaningful portion of the output stream to a file for later analysis, and they can't submit saved files as input.

- The target workstation must have previously installed the emulation software.

Client-Server Communications

Splitting an application into client and server pieces implies that the client and server can communicate: they must have a common communications medium and common protocols. For Macintosh clients AppleTalk™ is a reasonable protocol choice since it is built in to every Macintosh, but TCP/IP may also be a viable choice. TCP/IP is the most common protocol in the UNIX™ world. For DOS, Windows, and OS/2 clients, the NetWare protocol IPX/SPX is probably most common, with NetBIOS and TCP/IP in second and third place, respectively.

These protocols provide only the foundation for the required communications facility. Architecturally, the client and server don't send byte streams or packets back and forth; they send messages. A *message protocol* is therefore needed. System 7.0 provides *AppleEvents* on the Macintosh, which would serve nicely, but it is not currently supported on other platforms.[13] *Transport Independent Remote Procedure Call* (TIRPC)[14] may also be sufficient, but is not generally available. If neither of these is sufficient for your needs, you may have to devise and implement your own higher-level protocol.

Note that the physical medium and the transport protocol could be different for every client platform so

[11]by Farallon Computing, Inc., Emeryville, California.
[12]by Meridian Technology, Inc., a subsidiary of MicroCom, Inc.
[13]UserLand, Inc., in Palo Alto, California has recently announced support for a subset of AppleEvents on Macintosh System 6.0.x.
[14]See [ATT 90] for a definitive reference.

long as the server supports all the media and protocols. The message protocol can also be different, but the server is then required not just to support all of them, but also to translate between them.

Peer-to-Peer Communications

If your application needs to support peer-to-peer communications among your clients, your message protocol needs to support *discovering, identifying, and targeting* other clients, and your server model must support routing peer messages to the target client. Some sort of broadcast mechanism may also be desirable. As all clients are connected to the server, discovering other clients can be done by simply querying the server, but you may have to handle multiple servers in a network.

Single-Platform Development

The Distributed MVC architecture combines the Model-View-Controller paradigm with the Client-Server paradigm. It was designed to support applications that need to run on multiple client platforms, but it is still a useful architecture even for applications that will never to run on more than one platform. The conceptual separation of the user interface from the semantic model provided by MVC, and the flexible communications provided by the client-server separation are useful even on a single platform.

Limitations

- If your application has strict real-time requirements, the communication overhead and the unpredictable server response time may be intolerable.

- If your application is mostly user interface, you may find that the added complexity of the client-server model does not provide sufficient benefit.

- However, a very resource-intensive model could become a performance bottleneck if many clients share the same server; a distributed server model may be required to provide acceptable performance.

- If your target environment is an internetwork containing many servers, and concurrently shared data or peer-to-peer communications is very important, this architecture will not satisfy your needs unless you add server-to-server communications to the Server Model.

Summary

We have presented the Distributed MVC architecture, which supports building functionally identical applications running on different platforms. The architecture combines the Client-Server architecture with the Model-View-Controller paradigm to provide the advantages of both, and emphasizes the importance of the Conceptual Model.

Applications may take full advantage of whatever graphical interface is available on each platform, or they can have a plain textual interface, but the user should form the same conceptual model of how the program works in each case.

Distributed MVC provides strong support for functional (command) scripting which, if implemented, makes Quality Assurance much easier.

Applications which need to support concurrently shared data access from multiple disparate clients, with real-time display updates, can do so without an inordinate amount of work.

The separation of the user interface from the implementation of the functionality results in reduced documentation effort and the ability to increase parallelism in development. Providing the same look and feel on each platform is simplified because the conceptual model is shared.

While Distributed MVC is not a panacea and is not applicable to all product markets, this paradigm can provide substantial benefits for a large number of applications.

Acknowledgements

The Distributed MVC architecture developed from a number of design discussions with Elizabeth Brennan and Mike Russell. Robin Anderson joined them in reviewing this paper; I wish to thank all of them for their help.

Bibliography

[Aho 72] *The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing,* by Alfred V. Aho and Jeffrey D. Ullman, Prentice-Hall, 1972. ISBN 0-13-914556-7.

[Aho 86] *Compilers: Principles, Techniques, and Tools,* Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Addison-Wesley, 1986. ISBN 0-201-10088-6.

[ATT 90] *AT&T UNIX System V Release 4 Programmer's Guide: Networking Interfaces,* Prentice-Hall, 1990.

[Coa 91] *Object-Oriented Analysis,* by Peter Coad and Edward Yourdon, Yourdon Press/Prentice-Hall, 1991. ISBN 0-13-629981-4.

[Kra 88] *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System,* by Glenn Krasner and Stephen Pope, ParcPlace Systems, August 1988.

[Mey 90] *Object-oriented Software Construction,* by Bertrand Meyer, Prentice-Hall, 1990. ISBN 0-13-629049-3.

[Nor 88] *The Psychology of Everyday Things,* by Donald Norman, Basic Books, 1988. ISBN 0-465-06700-3.